Esercitazioni di Sistemi Operativi
Linux

Ing. Vincenzo Agate

vincenzo.agate@unipa.it

Università degli Studi di Palermo

C.d.L. in Ing. Gestionale e Informatica e Ing. Informatica e Telecomunicazioni

A.A. 2016/17

Programmazione di shell

Sommario degli argomenti

- Conoscere le man pages!
- Costruire blocchi di istruzioni
- Proprietà e autorizzazioni
- Filtri
- Scripting
- Utility su processi e thread

Le sezioni del manuale

man è il paginatore dei manuali del sistema. Di solito ognuno degli argomenti pagina dati a man è il nome di un programma, di un'utility o di una funzione.
Il man è suddiviso in sezioni, ognuna delle quali contiene tipi di pagine differenti.

Tabella: Le sezioni del man

1	Programmi eseguibili e comandi della shell
2	Chiamate al sistema (funzioni fornite dal kernel)
3	Chiamate alle librerie (funzioni all'interno delle librerie di sistema)
4	File speciali (di solito trovabili in /dev)
5	Formati dei file e convenzioni p.es. /etc/passwd
6	Giochi
7	Pacchetti di macro e convenzioni p.es. man(7), groff(7).
8	Comandi per l'amministrazione del sistema (solo per root)
9	Routine del kernel [Non standard]

UNIVERSITÀ DEGLI STUDI DI PALERMO

Le sezioni del manuale

Per accedere ad una sezione particolare del man:

`$ man [1-9]` *nomecomando*

Di seguito alcuni esempi:

- `$ man cat`
- `$ man 1 cat`
- `$ man 1 printf` (Es. `$ printf "val = %d\n" 10`)
- `$ man 3 printf` (ES. *printf("val = %d\n", 10);* del C)
- `$ man 2 chmod` (chiamate di sistema di basso livello fornite dal kernel)
- `$ man 4 NULL` (echo "prova" > /dev/null)
- `$ man 5 passwd` (formato di file particolari, convenzioni es. /etc/passwd)
- `$ man 8 shutdown`

Quando non conoscete il nome di un comando, ma sapete quello che fa potete usare:

`$ apropos` *parolachiave* (Es. `$ apropos list`)

UNIVERSITÀ DEGLI STUDI DI PALERMO

Altri metacaratteri della shell

- **#**: considera ciò che segue come un commento
  - `$ #prova di commento`
- **`var=val`**: assegna il valore **val** alla variabile **var**
  - `$ var=5`
- **`$var`**: restituisce il valore della variabile di shell **var**
  - `$ echo $var`
- **`${var}`**: come sopra, ma utile per migliorare la leggibilità
- **`\c`**: il carattere **c** inteso letteralmente
- **`` `...` ``**: **esegue** eventuali comandi contenuti tra gli apici inversi e li sostituisce con i relativi output
  - `` $ echo `date "+%Y-%m-%d"` ``

UNIVERSITÀ DEGLI STUDI DI PALERMO

Altri metacaratteri della shell

- `$(...)`: **esegue** eventuali comandi contenuti tra le parentesi e li sostituisce con i relativi output
  - `$ echo $(date "+%Y-%m-%d")`
- `'...'`: la stringa contenuta tra gli apici non viene interpretata dalla shell
  - `echo 'val $var'`
- `"..."`: solo alcuni metacaratteri vengono interpretati (in particolare: **$**, **** e gli apici inversi)
  - `echo "val $var"`
- operatori `;`, `&&` e `||`
  - Eseguire più comandi in sequenza: `$ sleep 4 ; echo "Hello"`
  - Eseguire i comandi solo se i precedenti hanno successo: `$ mkdir a && cd a`
  - Eseguire un comando solo se il precedente ha fallito:
    ```
    $ (ping -c 1 -w 5 www.google.com && echo 'tutto su') ||
    echo 'server giu'
    ```

UNIVERSITÀ DEGLI STUDI DI PALERMO

Flussi di input/output

Di default i comandi Linux prendono l'input da tastiera (standard input) e mandano l'output ed eventuali messaggi di errore su video (standard output, error).

Tabella: I tre flussi di Input/Output

Identificativo	Nome	Default
0	Standard input	Tastiera
1	Standard output	Terminale
2	Standard error	Terminale

UNIVERSITÀ DEGLI STUDI DI PALERMO

Redirezione dell'I/O

- I normali flussi di input/output ed errore possono essere rediretti
  - `command > filename`: memorizza l'output di **command** nel file **filename**
  - `command >> filename`: aggiunge l'output di **command** in coda al file **filename** (append)
  - `command < filename`: l'input di **command** è letto dal file **filename**
- Questi si possono combinare:
  - `ls > temp`
  - `wc -l < temp`
  - `&>` corrisponde a stdout + stderr
  - `1>` corrisponde a stdout, `2>` corrisponde a stderr

UNIVERSITÀ DEGLI STUDI DI PALERMO

Redirezione dell'I/O: Esempio

Osserviamo il contenuto di un programma in C denominato redirect_io.c:

```
#include <stdio.h>

int main(void)
{
    printf("Stringa stampata sullo standard output\n");
    fprintf(stderr, "Stringa stampata sullo standard error\n");
    return 0;
}
```

Proviamo a compilarlo con: `$ gcc redirect_io.c`
Proviamo ad eseguirlo nelle seguenti modalità ed osserviamo ciò che viene stampato a video e sul file:

- `./a.out`
- `./a.out > log.txt`
- `./a.out 2> log.txt`
- `./a.out &> log.txt`

UNIVERSITÀ DEGLI STUDI DI PALERMO

Redirezione dell'I/O

- Connette l'output di un programma all'input di un altro
- Operatore |
- L'esempio

  `ls > temp`

  `wc -l < temp`

  può essere scritto come `ls | wc -l`

Creazione dei link

- Hard link: `$ ln file hlink`
- Link simbolici: `$ ln -s file slink`
- Osserviamo le differenze: `ls -l`

```
vincenzo@vincenzo-VirtualBox:~/Dropbox/esercitationi_SO/link$ ls -l
totale 16
-rw-rw-r-- 1 vincenzo vincenzo 11 apr 12 06:12 file1.txt
-rw-rw-r-- 1 vincenzo vincenzo 21 apr 12 06:10 hlink1
lrwxrwxrwx 1 vincenzo vincenzo  9 apr 12 06:05 slink1 -> file1.txt
```

Figura: Hard link e symbolic link

UNIVERSITÀ DEGLI STUDI DI PALERMO

Permessi

- **`chmod` permessi files**: cambia i permessi associati ai file o cartelle indicati
- L'operazione può essere effettuata dal proprietario o root
- Schema ottale: 4=read, 2=write, 1=execute
- Shortcuts:
  - utenza: u(user), g(group), o(other), a(all)
  - azioni: r, w, x
  - tipo di permesso: +, -, =
- Setuid: **`chmod +s` files**: in esecuzione estende il permesso di chi possiede il file a tutti gli altri utenti (esempio: **`/usr/bin/passwd`**)
  Pericoloso per i file di root!

UNIVERSITÀ DEGLI STUDI DI PALERMO

Permessi - Esempi

- Notazione alfabetica:
  - `chmod ugo+rwx nomefile` # assegna ad user, group e other i permessi di lettura, scrittura ed esecuzione associati al file nomefile
  - `chmod ugo-rwx nomefile` # toglie ogni tipo di permesso a user, group e other
  - `chmod u+x nomefile` # assegna ad user il permesso di esecuzione associato al file nomefile
  - `chmod u-x nomefile` # toglie il permesso di esecuzione ad user
- Notazione numerica (indica direttamente i permessi da attribuire al file senza tener conto di quelli già presenti):
  - `chmod 777 nomefile` # assegna ad user, group e other i permessi di lettura, scrittura ed esecuzione associati al file nomefile
  - `chmod 000 nomefile` # toglie ogni tipo di permesso a user, group e other
  - `chmod 100 nomefile` # assegna solo ad user il solo permesso di esecuzione associato al file nomefile

UNIVERSITÀ DEGLI STUDI DI PALERMO

Cambiare utenza

- `su` **username**: cambia l'utente corrente. Se dato senza argomenti viene inteso l'utente root
- `chown` **username files**: cambia il proprietario dei file indicati
- `chgrp` **groupname files**: cambia il il gruppo dei file indicati

UNIVERSITÀ DEGLI STUDI DI PALERMO

Filtri

- Sono programmi che leggono qualche input, effettuano trasformazioni su di esso, e producono qualche output
  - `grep`
  - `sort`
  - `wc`
  - `...`
- Filtri programmabili
  - Le trasformazioni sono espresse sotto forma di un vero e proprio linguaggio di programmazione
  - I più diffusi sono **`awk`** e **`sed`**

(e)grep

- `grep` **pattern filenames**: stampa tutte le righe che contengono un'istanza di **pattern**
- Come creare il pattern?
- Espressioni Regolari:

`^`	inizio della riga
`$`	fine della riga
`.`	ogni carattere singolo
`[...]`	ogni carattere singolo in ... (anche intervalli)
`[^...]`	ogni carattere singolo **non** in ... (anche intervalli)
`r*`	zero o più occorrenze di **r**
`r+`	una o più occorrenze di **r**
`r?`	zero o una occorrenza di **r**
`r{n}`	esattamente **n** occorrenze di **r**

UNIVERSITÀ DEGLI STUDI DI PALERMO

(e)grep

- Alcuni esempi:
  - `ls -l | grep '^\-..x..x..x'`
    mostra tutti i file (nella directory corrente) eseguibili da owner, group e others
  - `grep '^r[a-z]*s$' /usr/share/dict/words`
    trova tutte le parole inglesi che iniziano per r e finiscono per s
    /usr/share/dict/words (o /usr/dict/words) è il dizionario standard di UNIX (Linux) con circa 480K parole
  - `grep '^[^:]*:' /etc/passwd`
    filtra il file delle utenze

awk

`awk` è un interprete per il linguaggio di programmazione AWK, un potente linguaggio utile alla manipolazione di dati e testi. Opera per righe di input (file o pipe).

- Utilizzo:

```
$ awk [-F value] [-v var=value] 'program text' [file...]
```

- Un programma awk è una sequenza di coppie ***pattern {action}***
  - pattern: espressioni regolari o condizioni espresse in linguaggio simile al C
  - action: sequenza di istruzioni con linguaggio simile al C (printf, for, while, ...)
- Entrambi gli elementi sono opzionali, ma non contemporaneamente:
  - `awk '/regex/' filename` - Per le righe individuate dall'espressione regolare l'azione predefinita è quella di stamparle
  - `awk '{action}' filename` - Le righe sono individuate tutte, ma viene eseguita l'azione specificata

awk

L'input di **awk**, è automaticamente suddiviso in campi, il cui separatore è di default lo spazio o il tab.

- Il separatore può essere specificato attraverso l'opzione:
  `awk -F nuovoCarattereSeparatore 'pattern {action}'`
- Per individuare ognuno dei campi: `$1, $2, ..., $NF`
- `$0` è l'intera riga: `ls -l | awk '{print $0}'`
- Per indicare il pattern deve essere soddisfatto su un campo specifico usare la tilde: `awk '$n ~ /regex/ {action}'`
- Esempio: `awk -F : 'length($2) == 0 {print $1}' /etc/passwd` mostra gli utenti la cui password non è impostata

UNIVERSITÀ DEGLI STUDI DI PALERMO

Esempi

```
$ ls -la | awk '{print $1}'
$ ls -la | awk '/^[-dl]/'
$ ls -la | awk '/^[-]/ {print $9}'
$ ls -la | awk '$8 ~ /^hh:mm/ {print $9}'} #hh:mm ora e
    minuto di modifica del file
$ ls -la | awk '$5 ~ /0$/ && $1 ~ /^-/ {print NF}' #è
    possibile applicare diverse espressioni regolari su
    diversi campi contemporaneamente
$ ls -la | awk '{for(i=1;i<=NF;i++) {var = var " " $i} print
    var}' #l'action accetta istruzioni in linguaggio simile
    al C
```

UNIVERSITÀ DEGLI STUDI DI PALERMO

Creazione di nuovi comandi

- Esempio: Supponiamo di voler frequentemente contare il numero di file presenti in una cartella
- Dovremmo scrivere qualcosa del genere:
  `ls -lA | grep ^[^dt] | wc -l`
- Invece di scriverlo ogni volta che ci serve, possiamo creare un programma di shell chiamandolo opportunamente:
  `echo 'ls -lA | grep ^[^dt] | wc -l' > cf`
- Se un file è eseguibile ed è un file di testo, e il testo è interpretabile dalla shell, diventa un comando:
  `chmod +x cf` (o anche solo `chmod u+x cf`)
- il comando potrà a questo punto essere richiamato: `./cf`

UNIVERSITÀ DEGLI STUDI DI PALERMO

Variabili d'ambiente

- Linux ricerca un programma eseguibile in determinate directory
- Il percorso di ricerca è memorizzato nella variabile `PATH`
  ```
  echo $PATH
  :/usr/local/bin:/usr/bin:/bin
  ```
- Il comando **set** mostra tutte le variabili d'ambiente
- Il valore di una variabile di ambiente può essere cambiato:
  ```
  PATH = .:$PATH
  ```
  (aggiunge la directory corrente)
- Il comando precedente funziona solo per l'utente e la shell corrente, per rendere attivo il cambiamento:
  ```
  export PATH
  ```
- Per rendere le modifiche permanenti dobbiamo modificare uno dei seguenti file: `.bashrc` e `.bash_profile`

UNIVERSITÀ DEGLI STUDI DI PALERMO

Programmazione della shell

- La shell offre un vero e proprio linguaggio di programmazione
- Creazione di file script
  - un normalissimo file di testo contiene una serie di istruzioni che possono essere eseguite da un interprete
  - occorre richiamare l'interprete: `(ba)sh scriptfile`
- Per evitare questa trafila la prima riga di ogni file di script deve contenere lo shabang **#!**:
  - **#!** percorso-assoluto-programma-interprete
    `#!/bin/bash`
  - occorre rendere eseguibile il file di script
    `chmod +x scriptfile`

Argomenti di input

- `$#` Numero di argomenti
- `$*, $@` Tutti gli argomenti passati alla shell
- `$1, $2, ...` Argomento di posto 1, 2, ...
- `$0` Nome del comando stesso
- `$-` Opzioni passate alla shell
- `$?` L'ultimo comando eseguito
- `$$` PID della shell

Riscrivere il programma conta file (cf), assicurandosi che possa ricevere qualsiasi directory come input...

UNIVERSITÀ DEGLI STUDI DI PALERMO

Esercizio

Data una directory con il seguente contenuto:

```
drwxr-xr-x 2 vincenzo  staff  68 11 Apr 20:01 dirA
drwxr-xr-x 2 vincenzo  staff  68 11 Apr 20:01 dirB
-rw-r--r-- 1 vincenzo  staff   0 11 Apr 20:00 file-a1
-rw-r--r-- 1 vincenzo  staff   0 11 Apr 20:00 file-a10
-rw-r--r-- 1 vincenzo  staff   0 11 Apr 20:00 file-a2
-rw-r--r-- 1 vincenzo  staff   0 11 Apr 20:00 file-b15
-rw-r--r-- 1 vincenzo  staff   0 11 Apr 20:00 file-b2
-rw-r--r-- 1 vincenzo  staff   0 11 Apr 20:00 file-b3
-rw-r--r-- 1 vincenzo  staff   0 11 Apr 20:00 file-c4
lrwxr-xr-x 1 vincenzo  staff   7 11 Apr 20:01 link-a2 -> file-a2
-rw-r--r-- 1 vincenzo  staff   0 11 Apr 20:00 prova
```

Si crei un comando che sposti i soli file/link il cui nome termina per lettera **a** seguita da un **numero** nella directory **dirA** e quelli il cui nome termina per lettera **b** seguita da un **numero** nella directory **dirB**.

UNIVERSITÀ DEGLI STUDI DI PALERMO

Soluzione

Una possibile soluzione:

```
#!/bin/bash
mv $(ls -l | awk '/^[-l].*a[0-9]+$/ {printf("%s ",$9)}') dirA
mv $(ls -l | awk '/^[-l].*b[0-9]+$/ {printf("%s ",$9)}') dirB
```

Comandi e sintassi - `for`

- Sintassi:

```
for variable [in valore]

do
    lista di comandi
done
```

- Esempio:

```
#!/bin/bash
for i in $@
do
    echo $i
done
```

UNIVERSITÀ DEGLI STUDI DI PALERMO

Comandi e sintassi - case

- Sintassi:

```
case expression in
    pattern1 )
        statements ;;
    pattern2 )
        statements ;;
    ...
esac
```

- Esempio

```
#!/bin/bash
case $1 in
    -a | -A | --aplha )
     echo "alpha" ;;
    -b )    echo "bravo"
    ;;
    -c )    echo "charlie"
     ;;
    * )    echo "opzione
    sconosciuta" ;;
esac
```

Comandi e sintassi - if

- Sintassi

```
if conditionlist
then
    lista comandi
[elif conditionlist
then
    lista comandi]
...
[else
    lista comandi]
fi
```

- Esempio:

```
#! /bin/bash
file=$1
if [ -e $file ]
then
  echo -e "File $file
    exists"
else
  echo -e "File $file
    doesnt exists"
fi
```

UNIVERSITÀ DEGLI STUDI DI PALERMO

Comandi e sintassi - `while`

- Sintassi

```
while conditionlist
do
    lista comandi
done
```

- Esempio

```
#!/bin/bash
counter=$1
factorial=1
while [ $counter -gt 0 ]
do
   factorial=$(( $factorial *
    $counter ))
   counter=$(( $counter - 1 ))
done
echo $factorial
```

UNIVERSITÀ DEGLI STUDI DI PALERMO

Funzioni

- Attraverso le funzioni è possibile dare un nome a un gruppo di comandi
- Si possono richiamare come si fa per un comando interno normale
- Le funzioni restituiscono un valore di uscita
  - il valore di uscita è solitamente un intero rappresentante lo stato (analogo al valore passato ad exit() in un processo)
  - il valore di uscita è memorizzato in $?
  - se non specificato da una apposita return, lo stato di uscita della funzione corrisponde allo stato dell'ultimo statement eseguito
- Sintassi:

```
[function] nome () {
    lista di comandi
}
```

UNIVERSITÀ DEGLI STUDI DI PALERMO

Espressioni aritmetiche

- È possibile espandere le espressioni aritmetiche usando la seguenti sintassi:
  `$((espressione))`
  che opera soltanto su interi

Test

- Test **expression** (oppure, in breve, **[expression]**)
- Stringhe intese come nomi di file:

`-r`	True if file exists and is readble
`-w`	True if file exists and is writable
`-x`	True if file exists and is executable
`-f`	True if file exists and is a regular file
`-d`	True if file exists and is a directory
`-c`	True if file exists and is a character special file
`-b`	True if file exists and is a block special file
`-p`	True if file exists and is a named pipe (FIFO)
`-u`	True if file exists and is a SETUID file
`-g`	True if file exists and is a SETGID file
`-k`	True if file exists and the sticky bit is set
`-s`	True if file exists and has a size grater than zero

Test

- Test **expression** (oppure, in breve, **[expression]**)
- Stringhe intese come interi:

`-ne`	Not equal
`-eq`	Equal
`-gt`	Greater then
`-ge`	Greater then or equal
`-le`	Less than or equal
`-lt`	Less than

Test

- Test **expression** (oppure, in breve, **[expression]**)
- Stringhe intese come interi:

  `-z str` (Not equal)
  `-n str` (Equal)
  `str1 = str2` (Greater then)
  `str1 != str2` (Greater then or equal)
  `str` (Less than or equal)
  `-lt` (Less than)

Comandi utili - Processi

- priorità e temporizzazione
  - nice
  - nice -n 10 sort input.txt > output.txt
  - nohup, at, cron
- tracciare utenti e binari
  - w, who, last
  - which
- altri metacaratteri della shell: &, &&, ||
  - &: eseguire un processo in background (anche: bg; e poi: fg)
  - &&, ||: AND e OR logici per i processi

Comandi utili - Threads

- Compilare in bytecode per la Java Virtual Machine
  - javac NomeClasse.java
  - java NomeClasse
- Compilare in codice oggetto
  - //se necessario possiamo definire un alias per il comando
  - alias gcj=gcj-4.8
  - gcj-main=NomeClasse NomeClasse.java -o nomeclasse
- Per avere informazioni sui thread
  - // mostrare tutti i thread di un processo java
  - `ps -eLf | grep java`
  - // mostrare tutti i thread dell'utente vincenzo, mostrando i LWP
  - `ps -u vincenzo -L -o pid,ppid,lwp,comm | grep java`
  - // mostrare tutti i thread di uno specifico terminale (es: pts/6), mostrando i LWP
  - `ps -t pts/6 -L -o pid,ppid,tty,lwp,comm`
  - // mostrare l'albero dei processi
  - `pstree`

Comandi utili - Threads

- Per ottenere il pid dell'applicazione java
  - `jps -V | grep` *`nomeclasse`*
- Per avere tutte le informazioni sui thread (compresi quelli della JVM) associati al pid appena trovato:
  - `jstack` *`pidNumber`* `| less`
- I comandi jps e jstack sono sperimentali e potrebbero non essere supportati da tutti i sistemi!

UNIVERSITÀ DEGLI STUDI DI PALERMO